

HANDOUT ASTEROIDS - AULA PYGAME

Os objetivos de aprendizado dessa aula são:

- Recriar um jogo simples baseado no Asteroids, usando o *framework* Pygame.

Resumo:

Asteroids é um jogo de nave que foi muito famoso na década de 80. Consiste em destruir os meteoros que vem em sua direção através de tiros. Nesse tutorial, vamos replicar uma versão simplificada do jogo usando o Pygame. Esse tutorial foi baseado no site Kidscancode, o qual possui mais tutoriais com mais técnicas de desenvolvimento de jogos.

O Pygame é um *Framework* (ou *game engine*) para desenvolvimento de jogos em Python, baseado na biblioteca SDL2 (Simple DirectMedia Layer). Para mais informações consulte o site do Pygame: <https://www.pygame.org>

1. INSTALAÇÃO DO PYGAME:

Utilize o comando *pip* do Python no **Anaconda prompt**, o qual baixará a última versão do pacote adequada ao seu sistema e a versão do Python que você tem instalada. Obviamente você precisará estar conectado à Internet para realizar esse procedimento.

```
pip install pygame
```

2. ESTRUTURA BÁSICA:

O nosso código terá as seguintes partes:

Classes: onde serão codificadas as classes Player, Mob e Bullet.

Inicialização, onde o pygame será iniciado, a tela principal do jogo criada e os objetos instanciados.

Looping Principal: onde tudo acontecerá.

3. PREPARANDO O JOGO:

Faça um download do arquivo assets.zip no Blackboard. Em seguida crie um repositório no seu GitHub e clone na sua máquina. Descompacte os arquivos nessa pasta, mantendo as pastas img e snd. Realize o *add*, *commit* e *push*.

Nome	Tipo
img	Pasta de arquivos
snd	Pasta de arquivos
asteroids-base	Arquivo PY

Abra no Spyder e rode o arquivo `asteroids-base.py`. Segue uma explicação:

Linhas 4 e 5: são importados os pacotes `pygame` e `os` (para poder usar comandos do módulo `path`, como `join`).

Linhas 8: cria uma string que representa o caminho (pasta) onde as imagens estão alocadas.

Linhas 11 a 13: estabelece o tamanho da tela e a velocidade de execução.

Linhas 16 a 21: define algumas cores básicas, no formato RGB.

Linha 24 e 25: o framework `Pygame` é iniciado. Apenas após esse comando (`pygame.init`) é que os recursos do pacote podem ser utilizados.

Linha 28: é criada uma janela (display surface) a partir do comando `display.set_mode`. O primeiro parâmetro é um par de coordenadas, que definem largura e altura da janela.

Linha 31: é definido o “caption” da janela (texto que aparecerá na barra superior da janela).

Linha 34 e 35: é carregada a imagem de fundo (o método `convert()` ajusta as cores da imagem para o padrão do `pygame`, garantindo maior eficiência).

Linha 42: Início do looping infinito que executará as atualizações do jogo repetidamente.

Linhas 45 a 49: para cada evento da lista de evento é verificado se um deles é `QUIT` (evento que é gerado quando o jogador fecha a janela). Se for, ajusta a variável para sair do loop.

Linhas 52 a 56: a cada passagem do loop, atualiza a tela e os sprites.

Linha 59: diz ao `pygame` para sair do jogo de forma suave.

PARTE 1: ADICIONANDO UMA NAVE

Digite o código a seguir, logo após as definições das cores:

```

23 # Classe Jogador que representa a nave
24 class Player(pygame.sprite.Sprite):
25
26     # Construtor da classe.
27     def __init__(self):
28
29         # Construtor da classe pai (Sprite).
30         pygame.sprite.Sprite.__init__(self)
31
32         # Carregando a imagem de fundo.
33         player_img = pygame.image.load(path.join(img_dir, "playerShip1_orange.png"))
34         self.image = player_img
35
36         # Diminuindo o tamanho da imagem.
37         self.image = pygame.transform.scale(player_img, (50, 38))
38
39         # Deixando transparente.
40         self.image.set_colorkey(BLACK)
41
42         # Detalhes sobre o posicionamento.
43         self.rect = self.image.get_rect()
44
45         # Centraliza embaixo da tela.
46         self.rect.centerx = WIDTH / 2
47         self.rect.bottom = HEIGHT - 10
48

```

Será codificada a **classe** Player, que tem como **atributos** a imagem e o retângulo representativo do objeto. Essa classe recebe os atributos e **métodos** de uma classe pré-existente chamada Sprite (pygame.sprite.Sprite) e isso é chamado de **Herança**.

A classe acima apenas define-a (como uma def). Adicione agora um **objeto** dessa nave pouco antes do **try**:

```

61 background_rect = background.get_rect()
62
63 # Cria uma nave. O construtor será chamado automaticamente.
64 player = Player()
65
66 # Cria um grupo de sprites e adiciona a nave.
67 all_sprites = pygame.sprite.Group()
68 all_sprites.add(player)
69
70 # Comando para evitar travamentos.
71 try:
72
73     # Loop principal.
74     running = True

```

E diga para o programa desenhar a nave:

```
85     screen.fill(BLACK)
86     screen.blit(background, background_rect)
87     all_sprites.draw(screen)
88
89     # Depois de desenhar tudo, inverte o display.
90     pygame.display.flip()
91
```

Finalizando: *commit*, *pull*, *push* no repositório.

PARTE 2: ADICIONANDO MOVIMENTO

Crie um atributo novo, chamado *speedx* no *__init__* da classe *Player*:

```
46     # Centraliza embaixo da tela.
47     self.rect.centerx = WIDTH / 2
48     self.rect.bottom = HEIGHT - 10
49
50     # Velocidade da nave
51     self.speedx = 0
```

Adicione a captura das setas no teclado no *for* de eventos:

```
150     # Verifica se foi fechado.
151     if event.type == pygame.QUIT:
152         running = False
153
154     # Verifica se apertou alguma tecla.
155     if event.type == pygame.KEYDOWN:
156         # Dependendo da tecla, altera a velocidade.
157         if event.key == pygame.K_LEFT:
158             player.speedx = -8
159         if event.key == pygame.K_RIGHT:
160             player.speedx = 8
161
162     # Verifica se soltou alguma tecla.
163     if event.type == pygame.KEYUP:
164         # Dependendo da tecla, altera a velocidade.
165         if event.key == pygame.K_LEFT:
166             player.speedx = 0
167         if event.key == pygame.K_RIGHT:
168             player.speedx = 0
```

Agora coloque o método *update* na classe *Player*:

```
46         # Centraliza embaixo da tela.
47         self.rect.centerx = WIDTH / 2
48         self.rect.bottom = HEIGHT - 10
49
50         # Velocidade da nave
51         self.speedx = 0
52
53     # Metodo que atualiza a posição da navinha
54     def update(self):
55         self.rect.x += self.speedx
56
57         # Mantem dentro da tela
58         if self.rect.right > WIDTH:
59             self.rect.right = WIDTH
60         if self.rect.left < 0:
61             self.rect.left = 0
62
```

Por último, atualizar o status da nave a cada loop (após processar os eventos):

```
162         # Verifica se soltou alguma tecla.
163         if event.type == pygame.KEYUP:
164             # Dependendo da tecla, altera a velocidade.
165             if event.key == pygame.K_LEFT:
166                 player.speedx = 0
167             if event.key == pygame.K_RIGHT:
168                 player.speedx = 0
169
170         # Depois de processar os eventos.
171         # Atualiza a acao de cada sprite.
172         all_sprites.update()
173
```

Rode. Percebeu que a navinha se movimenta muito rapidamente?

Isso acontece porque estamos em um looping infinito, cuja velocidade de execução depende da CPU, memória e outras características do equipamento onde estiver sendo executado. Dessa forma a velocidade de deslocamento da nave vai variar conforme a máquina, e os programas que estejam nela rodando, em que o jogo for executado.

Para controlar o número de iterações do looping por segundo usaremos o seguinte artifício. Será criado um relógio, a partir da classe `Clock` do Pygame. Essa classe possui um método chamado `tick` que conta quantos milissegundos já se passaram desde sua criação. Você já deve estar imaginando a utilidade desse recurso não? Mas por enquanto subutilizaremos esse relógio. Pegaremos o tempo transcorrido mas não utilizaremos para nada. Mas ao pegar esse tempo especificaremos para o método `tick` que queremos que ele seja executado apenas 30 vezes por segundo. Dessa forma teremos um efeito colateral útil: o looping será amarrado por essa chamada e só executará 30 vezes por segundo também. Assim, em qualquer máquina que o código for executado a velocidade de atualização do jogo será a mesma (exceto, claro, se a máquina for tão lenta que não conseguir executar nessa velocidade).

Insira uma variável `clock` antes de carregar o background:

```
111 # Nome do jogo
112 pygame.display.set_caption("Navinha")
113
114 # Variável para o ajuste de velocidade
115 clock = pygame.time.Clock()
116
117 # Carrega o fundo do jogo
118 background = pygame.image.load(path.join(img_dir, 'starfield.png')).convert()
119 background_rect = background.get_rect()
```

E dentro do Loop:

```
140 # Loop principal.
141 running = True
142 while running:
143
144     # Ajusta a velocidade do jogo.
145     clock.tick(FPS)
146
147     # Processa os eventos (mouse, teclado, botão, etc).
148     for event in pygame.event.get():
```

Agora é sua vez:

1. Crie uma classe chamada Mob que carrega a imagem do meteoro.
2. Coloque inicialmente o meteoro em uma posição aleatória em X entre [0,WIDTH] e em Y entre [-100,-40]. Usar `random.randrange()`.
3. Selecione uma velocidade aleatória em X (`speedx`) entre [-3, 3]
4. Selecione uma velocidade aleatória em Y (`speedy`) entre [2, 9]
5. Crie um grupo mobs, da mesma forma que criou `all_sprites`.
6. Após criar uma navinha, crie e adicione 8 objetos Mob no grupo mobs e no grupo `all_sprites`.

Finalizando a parte 2: *commit, pull, push* no repositório.

PARTE 3: COLISÃO

Nessa sessão vamos fazer a navinha morrer quando colidir com um meteoro. Se você colocou os Sprites (objetos) em grupos, fica muito fácil dizer para o pygame detectar as colisões. Antes vamos carregar alguns sons para usar no jogo:

1. Adicione a variável `snd_dir` que carrega o path até a pasta `snd`. Semelhante a `img`.
2. Carregue os sons em variáveis para utilizar no jogo:

```
125 # Carrega o fundo do jogo
126 background = pygame.image.load(path.join(img_dir, 'starfield.png')).convert()
127 background_rect = background.get_rect()
128
129 # Carrega os sons do jogo
130 pygame.mixer.music.load(path.join(snd_dir, 'tgfcoder-FrozenJam-SeamlessLoop.ogg'))
131 pygame.mixer.music.set_volume(0.4)
132 boom_sound = pygame.mixer.Sound(path.join(snd_dir, 'expl3.wav'))
133
134 # Cria uma nave. O construtor será chamado automaticamente.
135 player = Player()
```

3. Dê *play* no música de fundo antes do `while`:

```
153 # Loop principal.
154 pygame.mixer.music.play(loops=-1)
155 running = True
156 while running:
157
```

Agora vamos começar a detectar colisão:

```

184     # Depois de processar os eventos.
185     # Atualiza a acao de cada sprite.
186     all_sprites.update()
187
188     # Verifica se houve colisao entre nave e meteoro
189     hits = pygame.sprite.spritecollide(player, mobs, False, pygame.sprite.collide_circle)
190     if hits:
191         # Toca o som da colisao
192         boom_sound.play()
193         time.sleep(1) # Precisa esperar senao fecha
194
195         running = False
196
197     # A cada loop, redesenha o fundo e os sprites
198     screen.fill(BLACK)
199     screen.blit(background, background_rect)
200     all_sprites.draw(screen)

```

A variável hits vai indicar se houve alguma colisão entre os meteoros dentro do grupo mobs com a nave (player). Ainda vamos dizer que é para testar a colisão no formato de círculo que é mais realista que retângulo. E para isso precisamos fazer dois ajustes em cada classe para indicar o raio do círculo. Na classe Player:

```

52     # Velocidade da nave
53     self.speedx = 0
54
55     # Melhora a colisao estabelecendo um raio de um circulo
56     self.radius = 25
57
58     # Metodo que atualiza a posicao da navinha
59     def update(self):
60         self.rect.x += self.speedx
61
62         self.speedx = random.randrange(-3, 3)
63         self.speedy = random.randrange(2, 9)
64
65         # Melhora a colisao estabelecendo um raio de um circulo
66         self.radius = int(self.rect.width * .85 / 2)
67
68     # Metodo que atualiza a posicao da navinha
69     def update(self):
70         self.rect.x += self.speedx
71         self.rect.y += self.speedy

```

E na class e Mob :

Finalizando a parte 3: *commit*, *pull*, *push* no repositório.

PARTE 3: TIROS E FINALIZANDO O JOGO

Nessa parte não teremos nenhuma novidade, vamos colocar tiros e capturar a colisão.

1. Adicionar a classe `Bullet` que representa o tiro. Usar a figura `laserRed16.png` e iniciar com velocidade em `y` de `-10` (o tiro sobe).
2. Quando o jogador apertar o espaço, deve-se criar um tiro na posição onde está a nave naquele momento. Esse tiro deve ser adicionado a um novo grupo chamado `bullets` e também ao grupo `all_sprites`.
3. Tem que detectar a colisão entre `mobs` e `bullets`. Se isso ocorrer, remover os dois (colocar o terceiro argumento da função como `True`). Não se esqueça de recriar os meteoros que explodiram.
4. Carregar os outros dois sons (`expl6.wav` e `pew.wav`) para tocar quando destruir um meteoro e atirar, respectivamente.

Finalizando: *commit*, *pull*, *push* no repositório.

Pronto, agora vocês tem uma boa base de como começar um jogo em Pygame. A estrutura básica do Asteroids está pronta. Na próxima aula vamos adicionar mais detalhes que tornam o jogo mais interessante como: placar, vida, explosão, etc.

REFERÊNCIAS:

<http://www.progameempython.com.br/blog/tutorial-pygame-fazendo-jogos-com-python-parte-1/>

<https://www.pygame.org/docs/>

http://kidscancode.org/blog/2016/08/pygame_shmup_part_1/